3.3.3 Other `vector` Operations

We can access the elements of a `vector` the same way as we access the characters in a `string`:

vecEx1.cpp

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
    vector<int> v{ 1, 2, 3 };
    for (auto &i : v) // for each element in v (note: i is a reference)
        i *= i; // square the element value
    for (auto i : v) // for each element in v
        cout << i << " "; // print the element
    cout << endl;
    return 0;
}
```

Quick Check on Concept: What is the difference between `for (auto &i : v)` and `for (auto i : v)`?

```
for (auto &i : v)
```

We define our control variable, `i`, as a reference so that we can use `i` to assign new values to the elements in `v`.

```
for (auto i : v)
```

Control variable, `i`, is a copy of an element in `v`. Any change in `i` will not affect the elements in `v`. Thus, we use this kind of `range for` for **read only access** in a container.

Subscript Operator `[]`

We can obtain a given element in `vector` using the subscript operator `[]` as with `string`s. Subscripts for `vector` start at 0 (a typical C/C++ convention). For example, the previous code can now be modified as:

vecEx2.cpp

```
#include <iostream>
#include <vector>
```

```
using namespace std;

int main(){
    vector<int> v{ 1, 2, 3 };
    for (decltype(v.size()) idx = 0; idx != v.size(); ++idx){
        v[idx] = v[idx] * v[idx];
        cout << v[idx] << " ";
    }
    cout << endl;
    return 0;
}
```

Exercise 3.4 In-class Coding Exercise

Ex34.cpp

Define and initialize a vector with 10 elements of 1 and print the contents. Modify all the even elements in the vector to 0 and print the modified contents. A sample output looks like:

```
The original elements in the vector container are:  1 1 1 1 1 1 1 1 1 1
The modified elements in the vector container are:  0 1 0 1 0 1 0 1 0 1
```

(Answer)

3.4 Introducing Iterators

(see ppt)

Looping through containers

Subscript Operator `[]`

```
// reset all the elements in ivec to 0
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)
     ivec[ix] = 0;
```

C++11 way

```
// reset all the elements in ivec to 0
for (decltype(ivec.size()) ix = 0; ix != ivec.size(); ++ix)
     ivec[ix] = 0;
```

Iterator

```
// using iterators to reset all the elements in ivec to 0
for (vector<int>::iterator iter = ivec.begin(); iter != ivec.end();
++iter)
        *iter = 0;  // set element to which iter refers to 0
```

C++11 way

```
// using iterators to reset all the elements in ivec to 0
for (auto iter = ivec.begin(); iter != ivec.end(); ++iter)
        *iter = 0;  // set element to which iter refers to 0
```

const_iterator for reading but not writing to the elements in the container.

```
string word;
vector<string> text;
while (cin >> word) {
    text.push_back(word);
}
for (vector<string>::const_iterator iter = text.begin();
     iter != text.end(); ++iter)
     cout << *iter << endl;
```

Q: C++11 way?

A:

Exercise 3.5 In-class Coding Exercise

Ex35.cpp

Rewrite Exercise 3.4 using iterator. Fill up the blank.

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector<int> ivec(10, 1);
   cout << "The original elements in the vector container are: "
      << " ";


//(your codes)


   cout << endl;
   cout << "The modified elements in the vector container are: "
      << " ";



//(your codes)



   cout << endl;
   return 0;
}
```

Ans:

Key Concept: Generic Programming

Programmers coming to C++ from C or Java might be surprised that we used `!=` rather than `<` in our for loops. C++ programmers use `!=` as a matter of habit. They do so for the same reason that **they use iterators rather than subscripts**: This coding style applies equally well to various kinds of containers provided by the standard library.

As we will learn later, only a few standard library types, `vector` and `string` being among them, have the subscript operator. Similarly, **all of the library containers** have iterators that define the `==` and `!=` operators. Most of those iterators do not have the `<` operator. By routinely using iterators and !=, we don't have to worry about the precise type of container we're processing.

Iterator Operations

Standard iterators support only a few operations, which are listed in below (Table 3.6).

`*iter`	Returns a reference to the element denoted by the iterator `iter`.
`iter->mem`	Dereferences `iter` and fetches the member named `mem` from the underlying element. Equivalent to `(*iter).mem`.
`++iter`	Increments `iter` to refer to the next element in the container.
`--iter`	Decrements `iter` to refer to the previous element in the container.
`iter1 == iter2` `iter1 != iter2`	Compares two iterators for equality (inequality). Two iterators are equal if they denote the same element or if they are the off-the-end iterator for the same container.

Remark: We can compare two valid iterators using == or `!=`. Iterators are equal (1) if they denote the same element or (2) if they are both off-the-end iterators for the same container. Otherwise, they are unequal. For example, we can simply write a code fragment that will capitalize the first character of a string.

```
string s("some string");
if (s.begin() != s.end()) { // make sure s is not empty
    auto it = s.begin(); // it denotes the first character in s
    *it = toupper(*it); // make that character uppercase
}
```

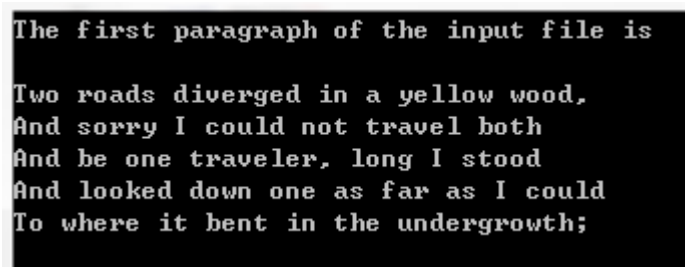
Exercise 3.6 In-class Coding Exercise

Ex36.cpp

A textfile `input.txt` contains sentences of text. A line with an empty string indicates a paragraph break. Write a program to store all the lines in a vector container and print the lines in the first paragraph. For example, if our `input.txt` has the following contents:

```
Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair,
And having perhaps the better claim
Because it was grassy and wanted wear,
Though as for that the passing there
Had worn them really about the same,

And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way
I doubted if I should ever come back.

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I,
I took the one less traveled by,
And that has made all the difference.

The Road Not Taken by Robert Frost
```

A sample output looks like:




```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    ifstream fin;
    fin.open("input.txt");
    if (!fin)
    {
        cerr << "cannot open input.txt" << endl;
        return -1;
    }
```

```
    string line;
    vector<string> text;
    while (getline(fin,line)) {
       text.push_back(line);
    }

    // print each line in text up to the first blank line




(your codes)




    return 0;
}
```

Answer:

```
    cout << "The first paragraph of the input file is" << endl << endl;
    for (auto it = text.cbegin(); it != text.cend()
       && !it->empty(); ++it)
       cout << *it << endl;
```

3.4.2. Iterator Arithmetic

Iterators for `string` and `vector` support additional operations that can move an iterator multiple elements at a time. They also support all the relational operators. These operations, which are often referred to as **iterator arithmetic**, are described below (Table 3.7).

`iter + n` `iter - n`	Adding (subtracting) an integral value `n` to (from) an iterator yields an iterator that many elements forward (backward) within the container. The resulting iterator must denote elements in, or one past the end of, the same container.
`iter1 += n` `iter1 -= n`	Compound-assignment for iterator addition and subtraction. Assigns to `iter1` the value of adding `n` to, or subtracting `n` from, `iter1`.
`iter1 - iter2`	Subtracting two iterators yields the number that when added to the right-hand iterator yields the left-hand iterator. The iterators must denote elements in, or one past the end of, the same container.
`>, >=, <, <=`	Relational operators on iterators. One iterator is less than another if it refers to an element that appears in the container before the one referred to by the other iterator. The iterators must denote elements in, or one past the end of, the same container.

3.5 Array

(see ppt)

The **general form** for declaring an array is:

```
typeName arrayName[arraySize];
```

The expression `arraySize`, which is the number of elements, must be a constant expression (must be known at compile time).

```
int arr[10]; // array of ten ints
//
unsigned cnt = 42; // not a constant expression
string bad[cnt]; // error: cnt is not a constant expression
//
const unsigned s = 42; // constant expression
int *parr[s]; // array of 42 pointers to int
```

Initialization (see ppt)

3.5.2. Accessing the Elements of an Array

As with the library `vector` and `string` types, we can use a `range for` or the subscript operator `[]` to access elements of an array.

When we use a variable to subscript an array, we normally should define that variable to have type `size_t`. `size_t` is a machine-specific unsigned type that is guaranteed to be large enough to hold the size of any object in memory. The `size_t` type is defined in the `cstddef` header, which is the C++ version of the `stddef.h` header from the C library. (you often do not need to explicitly include `cstddef` since many header files are likely included it already)

Quick Check on Concept: When we use a variable to subscript a string or a vector, we should define the variable using a machine-independent companion type, `string::size_type` and `vector<T>::size_type`.

(Example, see ppt)

3.5.3. Pointers and Array

Pointers Are Iterators

Pointers to array elements support the same operations as iterators on `vector`s or `string`s. For example, we can use the increment operator to move from one element in an array to the next:

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
int *p = arr; // p points to the first element in arr
++p; // p points to arr[1]
```

(C++11) Library begin and end functions

To make it easier and safer to use pointers, the new library includes two functions, named `begin` and `end`. These functions are defined in `<iterator>` and act like the similarly named container members.

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
int *beg = begin(ia); // pointer to the first element in ia
int *last = end(ia); // pointer one past the last element in ia
```

Quick Check: how to do the same thing in `vector`?

A:

Using `begin` and `end`, it is rather easy to write a loop to process the elements in an array.

```
int arr[20];
int *pbeg = begin(arr), *pend = end(arr);
while (pbeg != pend){
    //do something
    ++pbeg;
}
```

Quick Check: rewrite the following code using `begin` and `end` functions

arrayEx.cpp

```
int main(){
    const size_t array_size = 10;
    int ia[array_size];
```

```
    for (size_t ix = 0; ix < array_size; ++ix)
       ia[ix] = ix;
    return 0;
}
```

A:

arrayExIter.cpp